International Journal Of Advanced Research In Engineering Technology & Sciences ISSN: 2394-2819

Email: editor@ijarets.org February- 2015 Volume 2, Issue-2 www.ijarets.org

Fault Localization for Cots Components Using Dynamic Analysis Technique

T. Manju, C.Santhiya
Assistant Professor
Thiagarajar College of Engineering
Madurai

ABSTRACT
Fault localization is a process to find the location of faults. It determines the root cause of the failure. It identifies the causes of abnormal behaviour of a faulty program. It identifies exactly where the bugs are. Many software components are provided with incomplete specifications and little access to the source code. Reusing such gray-box components can result in integration faults that can be difficult to diagnose and locate. In this paper, we present Behavior Capture and Test (BCT), a technique that uses dynamic analysis to automatically identify the causes of failures and locate the related faults. BCT augments dynamic analysis techniques with model-based monitoring. In this way, BCT identifies a structured set of interactions and data values that are likely related to failures (failure causes), and indicates the components and the operations that are likely responsible for failures (fault locations). BCT advances scientific knowledge in several ways. It combines classic dynamic analysis with incremental finite state generation techniques to produce dynamic models that capture complementary aspects of component interactions. It uses an effective technique to filter false positives to reduce the effort of the analysis of the produced data. It defines a strategy to extract information about likely causes of failures by automatically ranking and relating the detected anomalies so that developers can focus their attention on the faults. The effectiveness of BCT depends on the quality of the dynamic models extracted from the program. BCT is particularly effective when the test cases sample the execution space well.

Keywords— Behavior Capture Test (BCT), Fault Localization, COTS components, Object Flattener tool

1. INTRODUCTION

Software systems often include components, hereafter gray-box components that are available with poor specifications and only give a partial view of the internal details. For example, many vendors offer Off-The-Shelf (OTS) components that generate complex reports from various data sources. Many of these OTS components and plug-in are offered without source code and with informal, incomplete specifications. Lack of access to source code and incomplete specifications harms the integration of gray-box components and can lead to integration faults with unpredictable effects.

Static analysis techniques can identify and remove some faults, but require access to the source code or need formal specifications, and generate many false positives (FP) that reduce their practical applicability[6], [7], [8].

Dynamic analysis produces information that can help developers identify and localize faults, even in the presence of gray-box components. When analyzing gray-box components,
dynamic analysis techniques monitor system executions by observing interactions at the component interface level, derive models of the expected behavior from the observed events, and mark the model violations, hereafter anomalies, as symptoms of faults [16], [17].

This paper proposes a solution, called Behavior Capture and Test (BCT), to analyze failures and diagnose faults. BCT is based on incremental dynamic analysis techniques that extract useful information without expensive storage consumption, multiple model violations, and reduces the impact of false positives.

The rest of this paper is organized as follows. Section2 discusses related work. Section3 presents the fault localization technique. Section 4 presents the details of our approach. Section 5 presents empirical results of our approach. Section 6 concludes.

2. RELATED WORK

FindBugs [13] is a bug pattern detector for Java. Find-Bugs uses a series of ad-hoc techniques designed to balance precision, efficiency, and usability. One of the main techniques FindBugs uses is to syntactically match source code to known suspicious programming practice, in a manner
similar to ASTLog [7]. For example, FindBugs checks that calls to wait(), used in multi-threaded Java programs, are always within a loop—which is the correct usage in most cases. In some cases, FindBugs also uses dataflow analysis to check for bugs. For example, FindBugs uses a simple, intra-procedural (within one method) dataflow analysis to check for null pointer dereferences. FindBugs can be expanded by writing custom bug detectors in Java. We set FindBugs to report "medium" priority warnings, which is the recommended setting.

Fault Localization is also done in software. Existing works represents fault localization in C, C++ and PHP Programs. Our previous work is Fault Localization for Java programs using Probabilistic Program Dependence Graph[10]. It scans each statement in Java program and it produces program dependence graph. To find the suspicious node ranking of nodes has done. For ranking probabilistic program dependence graph is generated. In this graph, the node having the least probability is considered to be most suspicious node.

3. BEHAVIOR CAPTURE TEST (BCT)

BCT is a technique that builds models of component interactions from successful executions of software systems and uses these models to analyze failures and diagnose faults. Successful executions are executions that satisfy the oracles or the user expectation, while failing executions are executions that are either aborted before normal termination (program crashes) or that violate the oracles (test failures) or the user expectation (field failures). BCT builds I/O and interaction models. I/O models are Boolean expressions that constraint the values that can be assigned to parameters. Interaction models are finite state automata (FSA) that specify sequences of methods that can be invoked by a component when interacting with other components, like the FSA in Fig. 1.

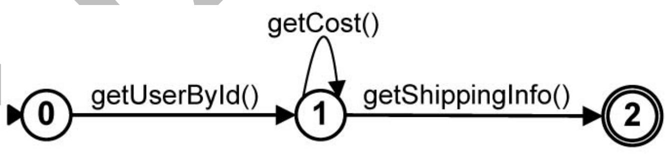


Fig. 1 The FSA associated with method order (Cart cart).

4. OUR APPROACH

BCT builds models of component interactions by dynamically analyzing system executions and monitors interactions at the component interface level, without accessing the implementation details of the components. Monitoring the internal of components would cause a high overhead and produce huge traces making dynamic model generation infeasible for nontrivial systems. BCT can be implemented on top of several technologies that range from aspect oriented programming to software probes. We experimented BCT with Microsoft Word

. BCT builds models that represent component interaction sequences within a single execution flow. When monitoring concurrent systems, BCT distinguishes invocation sequences that are part of the same execution flow from interleaving that depend on the concurrent structure by separately recording sequences of method calls observed in different threads.
To analyze failures and diagnose faults, BCT needs models that approximate the correct behavior of software systems. BCT builds accurate models of correct behaviors by monitoring software systems when executed thoroughly and successfully. Natural scenarios to produce such models are system, regression, and acceptance testing, when systems are executed thoroughly, and oracles prune failed executions.

BCT infers models that summarize and generalize the observed behaviors incrementally. In this way, BCT avoids recording a large amount of execution traces and improves scalability. BCT generates two types of models that we refer to as I/O and interaction models. I/O models are Boolean expressions that are associated with the methods in the component interfaces and that generalize the relations among values exchanged during the software executions. For example, an I/O model item: quantity > 0 associated with a method Cart. add(Item item) specifies that the attribute quantity of parameter item passed to method add implemented by the class Cart held only positive values in the monitored executions.

BCT generates I/O models with Daikon, an inference engine that can process data incrementally [32]. Interaction models are finite state automata that are associated with the methods in the component interfaces. They summarize the inter-component invocations involved in the method execution. An interaction model associated with a method m() implemented by a component C indicates the methods of the other components of the system that C invokes when m() is executed. For example, the FSA shown in Fig. 1 indicates the sequences of method invocations that have been observed while monitoring the component interactions of method order (Cart cart) executed to create new orders: Method order extracts user information by calling get User By ID() and retrieves the cost of all items that are part of the order (method get Cost()) before extracting the shipping address (method get Shipping Info ()). BCT infers interaction models using kBehavior.

4.1 Analyzing Failures and Diagnosing Faults
BCT analyzes failures and diagnoses faults by comparing failed executions with I/O and interaction models built during successful executions. In particular, BCT identifies unexpected parameter values that violate I/O models and unexpected method calls that violate interaction models, filters false positives by exploiting suitable heuristics, and clusters the resulting events to diagnose the possible faults. BCT compares failed executions with I/O and interaction models by reexecuting the software after a failure occurrence. To reproduce failures that occur during testing, test designers can simply reexecute the test cases that led to failures. Reproducing failures experienced in the field may be difficult due to the lack of execution details. To overcome these problems, it is possible to augment applications with frameworks that capture the runtime data necessary to repeat executions [33]. BCT does not depend on the implementation environment, except for monitoring. In this paper, we report our experience with Java, but BCT can be extended to programs written with other languages by substituting the monitoring framework.

4.2 kBehavior
A Nondeterministic Finite State Automaton is a five-tuple
$(Q, \Sigma, \partial, q0, F)$ Where

- Q is a nonempty finite set of states,
- Σ is a nonempty finite set of input symbols or input alphabet,
- δ:Q×∑->℘(Q) is the transition function that maps pairs of <state, symbol> into subsets of states,

- $q0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states.

With the trace generated by the object flattener tool, FSA is generated. kBehavior algorithm is used to generate the FSA. The FSA generated with our algorithm generates behaviors that suitably represent possible interactions among components reducing over generalization and over restrictiveness of other inference algorithms.

4.3 Design

The proposed technique is dynamic analysis technique. The dynamic analysis technique used is BCT. The framework for fault localization using BCT is shown in Figure5.1. It involves the following activities for fault localization. The design encompasses Interaction recording, FSA generation and Fault Localization.

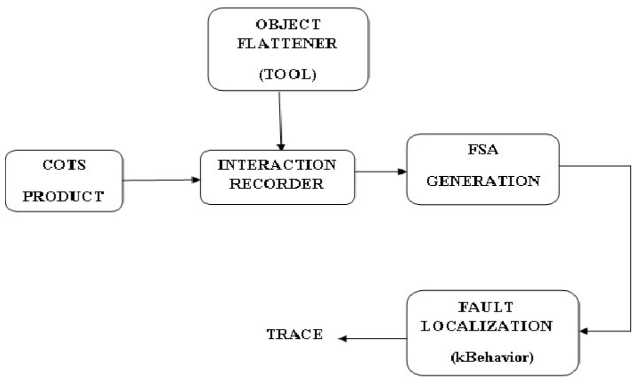


Fig. 2 Overall Design of Fault Localization

4.3.1 COTS Product

COTS (Commercial Off-the Shelf) Product [19] is a software product available in the market. COTS products are often only partially documented and developers may misuse technologies and introduce integration faults, as witnessed by the many entries in fault repositories. Once identified, common integration problems and their fixes are usually documented in forums and fault repositories on the Web, but this does not prevent them to occur in the field when COTS products are reused. COTS frameworks and components are reliable products, but the complexity of the technology and the incompleteness of the available documentation can result in faulty integration of COTS products. Some of the COTS components are MySQL, Oracle, Access, Word, MS Speech Engine, JSAPI, eSpeak, Festival, MacinTalk, SQLLite, SQLCompact, H2, SAPI TTS, JSML, SSML, STML, SABLE, AURAL CSS. In this work, Word and MySQL are taken into consideration.

4.3.2 Interaction Recorder

Interaction recorders extract information about component interactions. They store the beginning and the end of the executions of the monitored methods in a trace file. Methods are often invoked with references to complex objects as parameters. Simply recording the references would provide little information for generating invariants. To record useful information, we need to gain information about the state of the objects referred to by the parameters. BCT automatically extracts state information using the objects interfaces, therefore it can be applied when source code is not available and also to objects of languages different from Java. Classic approaches, like directly encoding the internal contents of objects with special inspectors, require access to the source code that may not be available when reusing components, thus a tool named "Object Flattener" is used. Object Flattener consists of identifying inspectors, i.e., public fields and methods that return state information

without altering the object. Object flattening recursively extracts state information until it obtains a primitive value, reaches a reference already inspected, or reaches a maximum depth. It is a prototype tool that is available at www.lta.disco.unimib.it/objectflattener
The prototype object flattener supports several models and formats for the extracted data, and provides extension mechanisms to cope with additional models or formats. The flattener can be extended also with plug-ins that implements specific strategies to select and invoke inspectors of particular sets of classes. Plug-ins is used to analyze objects of particular importance for the target application.

4.3.3 FSA Generation

With the trace generated by the object flattener tool, FSA is generated. KBehavior algorithm is used to generate the FSA. The FSA generated with our algorithm generates behaviors that suitably represent possible interactions among components reducing over generalization and over restrictiveness of other inference algorithms.

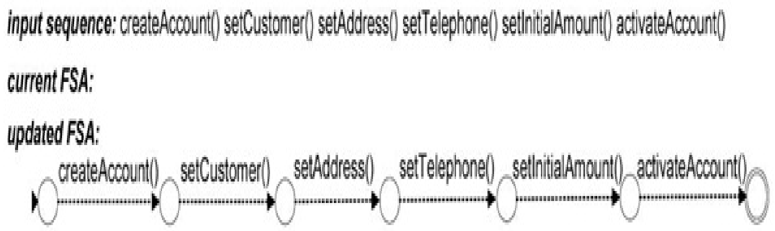


Fig. 3 The FSA that kBehavior generates by processing seq1.

Consider the following example that consists of three invocation sequences obtained by recording the interactions between a client component and a bank account manager component:

seq1:createAccount()setCustomer()setAddress()setTelephone
()setInitialAmount()activateAccount()
seq2:createAccount()setCustomer()setAddress()setMobile
Phone()setInitialAmount()activateAccount()
seq3:createAccount()setCustomer()setAddress()setMobile
Phone()addCustomer()addCustomer()addCustomer()addCustomer()setInitialAmount()activateAccount()
KBehavior processes the set of sequences incrementally, one after the other, without re-accessing sequences once processed. It starts with the first sequence (seq1) and builds an initial FSA that accepts the sequence. In some cases, the initial FSA maps the method calls to a linear sequence of transitions.

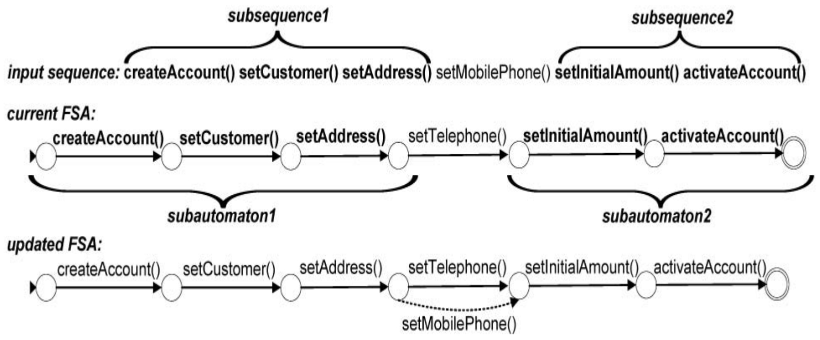


Fig. 4 The FSA that kBehavior generates by processing seq2 after seq1.

kBehavior processes the subsequences of the current sequence recursively, to produce a compact FSA. As an example, Fig. 3 shows how kBehavior processes seq3. kBehavior identifies the subautomata subautomaton1 that accepts the subsequence subsequence1 and subautomaton2 that accepts subsequence2, and connects the two subautomata with transitions that accept the four consecutive invocations of method addCustomer().

Before connecting subautomaton1 to subautomaton2, kBehavior processes the subsequence with the four invocations to addCustomer() recursively and produces the looping automaton.

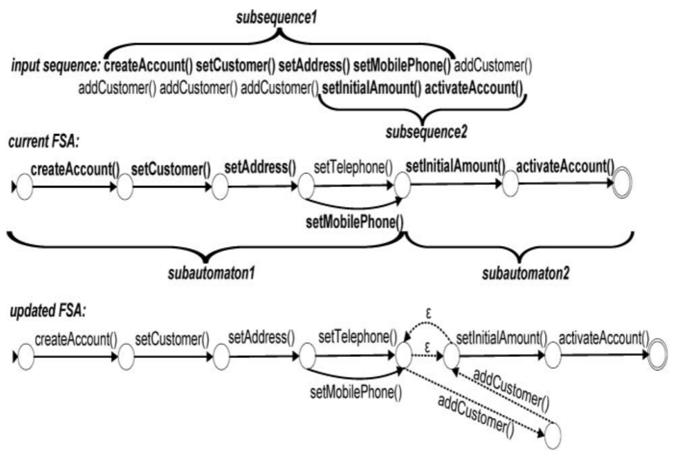


Fig.5 The FSA that kBehavior generates by processing seq3 after seq2 and seq1.

4.3.4 Fault Localization

BCT automatically distinguish violations that are likely related to faults from false positives with a simple but effective heuristic: Model violations that occur both during successful and failed executions are likely related to new software behaviors, while model violations that occur only during failed executions are most likely anomalies. Thus it automatically discard model violations that occur during both successful and failed executions, and analyzes violations that occur only during failed executions. This heuristic was effective in filtering false positives in our case studies.

The heuristic is inspired by the fault localization techniques proposed by Liu and Han [21] and Liblit et al. [22]. This heuristic requires the availability of both failing and successful executions. These are available, for example, when BCT investigates regression faults. Executing a regression test suite to validate a new version of a software system results in both failing and successful executions, and both kinds of executions can contain model violations that may derive either from faults (in failing executions only) or from changes implemented by developers (usually both in failing and successful ones). BCT analyzes failing executions, and uses successful executions to identify false positives.

Fault Localization is enhanced by kBehavior algorithm. If the
trace recorded by the object flattener tool is the expected trace, then it is termed as true trace and if the trace is not the expected trace, then it is termed to be fault trace. The fault trace is localized by comparing the expected

trace with the user given input and finds the location where it fails. Thus the exact location of the fault is known. Model based technique (FSA) is used because it is used to easily identify the fault.

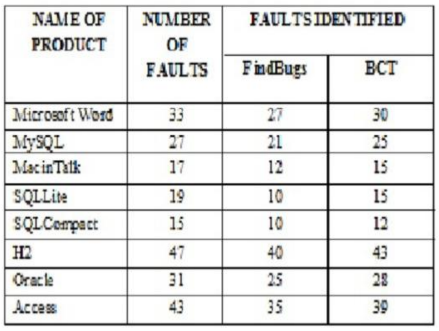
5. Experimental Evaluation

5.1 Results

This project tells about the effectiveness of the localization strategies by evaluating their fault detection rate. A product under test can be assessed by counting and classifying the discovered faults. In this work 8 COTS products are used and have generated traces for calculating the effectiveness of the fault localization. They are Microsoft Word, MySQL, MacinTalk, SQLLite, SQLCompact, H2, Oracle and Access. The number of faults detected is recorded for each product.

The comparison is made between the statistical and dynamic analysis technique. The statistical technique used is FindBugs and the dynamic analysis technique used is the proposed technique, BCT. The number of faults identified is comparatively large when compared to the FindBugs technique. The Comparative table is shown in Table.7.1

Table 1: Performance analysis



NAME OF PRODUCT	NUMBER OF FAULTS	FAULTS IDENTIFIED	
		FindBugs	BCT
Microsoft Word	33	27	30
MySQL	27	21	25
MacinTalk	17	12	15
SQLLite	19	10	15
SQLCompact	15	10	12
H2	47	40	43
Oracle	31	25	28
Access	43	35	39

From this comparative table, the number of faults identified by the BCT technique is high is shown clearly. So this is the efficient dynamic fault localization technique. Since it works without the source code. It is suitable for black box testing.

5.2 Performance Evaluation

The Performance Analysis measure is shown in the fig 6.1. Consider for example a COTS product Microsoft Word, the number of faults identified by FindBugs is 27. But for the same product BCT identifies 30 faults. So that we can confirm that BCT is an efficient technique than FindBugs.

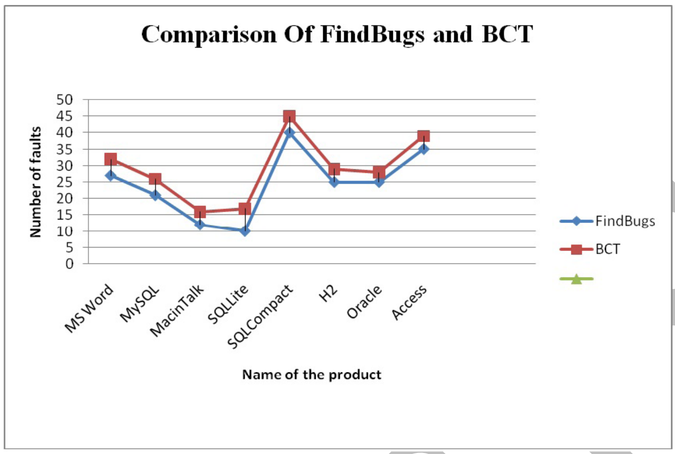


Fig. 6 Comparison Of FindBugs and BCT

6. CONCLUSION

This project proposes an dynamic analysis fault localization technique. This presents an innovative model for any COTS product. It scans each and every interactions of the components within the COTS product. The traces are designed as Finite State Automata(FSA). For FSA generation kBehavior algorithm is used and fault localization is done with the same algorithm. BCT analyzes the interactions of reused software components.

The technique collects information about the interactions of components when used in existing products, and uses the collected information to identify anomalous interactions of the components when they are updated or reused in new products. Anomalous interactions are behaviors not previously experienced and may be due to either new (legal) uses of the components, or erroneous interactions. BCT indicates functionalities that have not been fully tested yet and it signals faults and provides useful information for their localization.

7. FUTURE ENHANCEMENTS

In this work, new algorithms can be defined which works for both white-box and black-box testing. This work also shows that kBehavior algorithm can be accurate, depending on the context associated with the fault. In practice, it will be beneficial to harness the effectiveness of localizing approaches.

8. REFERENCES

1. Leonardo Mariani, Fabrizio Pastore and Mauro Pezze, "Dynamic Analysis for Diagnosing Integration Faults", IEEE Transactions On Software Engineering, vol. 37, NO. 4, July/August 2011.
2. Object Flattener, http://www.lta.disco.unimib.it/objectflattener.
3. J. Jones, M. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," Proc. 24th Int'l Conf. Software Eng., pp. 467-477, 2002.
4. M. Morisio, C.B. Seaman, A.T. Parra, V.R. Basili, S.E. Kraft, and S.E. Condon, "Investigating and Improving a COTS-Based Software Development," Proc. 22nd Int'l Conf. Software Eng., pp. 32-41, 2000.
5. Eric Wong, Vidroha Debroy, " SOFTWARE FAULT LOCALIZATION" IEEE Annual Technology Report, 2009.
6. D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," Proc. 19th Conf. Object-Oriented Programming Systems, Languages, and Applications, pp. 92-106, 2004.
7. N. Rutar, C.B. Almazan, and J.S. Foster, "A Comparison of Bug Finding Tools for Java," Proc. IEEE 15th Int'l Symp. Software Reliability Eng., pp. 245-256, 2004.
8. M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools Using Exploitable Bu_er Overows from Open Source Code," Proc. 12th Int'l Symp. Foundations of Software Eng., pp. 97-106, 2004.
9. S. Hissam and D. Carney, "Isolating Faults in Complex COTS Based Systems," white paper, Carnegie Mellon Software Eng. Inst., 1998.

10. A. Askarunisa, T. Manju and B. Giri Babu, " Fault Localization for Java Programs Using Probabilistic Program Dependence Graph", IJCSI International Journal of Computer Science Issues, Vol. 8, Issue 6, No 2, November 2011.
11. M. Renieris and S. Reiss, "Fault Localization with Nearest Neighbor Queries," Proc. IEEE 18th Int'l Conf. Automated Software Eng., pp. 30-39, 2003.
12. L. Briand, Y. Labiche, and X. Liu, "Using Machine Learning to Support Debugging with Tarantula," Proc. IEEE 18th Int'l Symp. Software Reliability Eng., pp. 137-146, 2007.
13. W. Masri, A. Podgurski, and D. Leon, "An Empirical Study of Test Case Filtering Techniques Based on Exercising Information Flows," IEEE Trans. Software Eng., vol. 33, no. 7, pp. 454-477, July 2007.
14. L. Mariani and M. Pezze`, "Behavior Capture and Test: Automated Analysis of Component Integration," Proc. IEEE 10th Int'l Conf. Eng. Complex Computer Systems, pp. 292-301, 2005.Journal of Automatic soft- ware Engineering, August 2009.
15. C. Liu and J. Han, "Failure Proximity: A Fault Localization-Based Approach," Proc. 14th Int'l Symp. Foundations of Software Eng., pp. 101- 112, 2006.
16. S. Hangal and M.S. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," Proc. 24th Int'l Conf. Software Eng., pp. 291-301, 2002.
17. A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting Object Usage Anomalies," Proc. Joint Meeting European Software Eng. Conf. and Symp. Foundations of Software Eng., 2007.
18. O. Raz, P. Koopman, and M. Shaw, "Semantic Anomaly Detection in Online Data Sources," Proc. 24th Int'l Conf. Software Eng., pp. 302-312, 2002.
19. L. Mariani and M. Pezze` , "Dynamic Detection of COTS Components Incompatibility," IEEE Software, vol. 24, no. 5, pp. 76-85, Sept./Oct. 2007.
20. SAP"CrystalReport",http://www.sap.com/solutions/sapbusinessobjects/sme/reporting/crystalreports/, 2009.
21. Herve Chang and Leonardo Mariani and Mauro Pezze," In-Field Healing of Integration Problems with COTS Components" ICSE'09, May 16-24, 2009.
22. Amandeep Kaur Johar and Shivani Goel, "COTS Components Usage Risks In Component Based Software Development", International Journal of Information Technology and Knowledge Management, Volume 4, No. 2, pp. 573-575, July-December 2011.
23. J.W. Nimmer and M. D. Ernst. "Static verification of dynamically detected program invariants: Integrating daikon and ESC/Java". In 1st Workshop on Runtime Verification, 2001.
24. L. Mariani, M. Pezze, and D. Willmor. "Generation of integration tests for self-testing components". In proceedings of the 1st International Workshop on Integration of Testing Methodologies, volume 3236 of LNCS, pages 337{350. Springer, 2004.
25. A. Biermann and J. Feldman. "On the synthesis of finite state machines from samples of their behavior". IEEE Transactions on Computer, 21:592{597, June 1972.
26. Andrews, J.H., Briand, L.C., Labiche, Y."Is Mutation an Appropriate Tool for Testing Experiments?" In: ICSE 2005, ACM, New York (2005).
27. Offut, A.J., Huffman- Hayes, J. "A Semantic Model of Program Faults". In: Proceedings of ISSTA, pp. 195{200 (1996).
28. Howden, W.E. "Reliability of the path analysis testing strategy". IEEE Trans. on Software Engineering 2(3), 208{215 (1976).
29. Eldh, S., Hansson, H., Punnekkat, S., Pettersson, A., Sundmark, D. "A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques". In: Proc. TAIC, IEEE, New York (2006).
30. Basili, V.R., Selby, R.W. "Comparing the Effectiveness of Software Testing Strategies original 1985", revised dec. 87. In: Boehm, B., Rombach, H.D., Zelkowitz, M.V. (eds.) Foundations of Empirical Software Engineering, The Legacy of Victor R. Basili, Springer, Heidelberg (2005)
31. .S. Hangal and M. S. Lam. "Tracking down software bugs using automatic anomaly detection". In Proceedings of the 24th International Conference on Software Engineering, pages 291{301, 2002.
32. M. J. Harrold, G. Rothermel, K. Sayre, R.Wu, and L. Yi. "An empirical investigation of the relationship between spectra differences and regression faults. Software Testing, Veri_cation and Reliability", 10(3):171{194,2000.
33. S. Horwitz and T. Reps. "The use of program dependence graphs in software engineering". In Proceedings of the 14th International Conference on Software Engineering, pages 392{411, May 1992.
34. J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In Proceedings of the 24th International Conference on Software Engineering, May 2002.
35. A. Zeller, "Why Programs Fail: A Guide to Systematic Debugging", Morgan Kaufman. 2005.